Bujar Sefa
CSC 21200 - BC
Professor Liu
HW #5

<u>Overview of homework:</u>
This homework is about creating a binary tree node class, binary tree class in both a dynamic array implementation and linked list implementation, a binary search tree, and heaps and heapsort. I had to create the header file which contained the prototypes of the function each class needed; binary tree functions include but are not limited to shift left, shift right, add left, and add rIght. Also, I had to create the implementation files for each function, i.e., for the size function, return the "space" (number of array spots) that was used — many of the classes interlinked with one another, for example, the binary search tree utilized binary tree node class and binary tree class.

<u>btNode</u>:
<u>Header File:</u> Sefa_Bujar_HW5_Q1.h:
```
#ifndef __BTNODE_H__
#define __BTNODE_H__

#include <cstdlib>
#include <cassert>
#include <iostream>
using namespace std;

template<class Item>
class btNode{
       public:
              btNode( const Item& init_data = Item(),
                            btNode<Item>* init_left = NULL,
                            btNode<Item>* init_right = NULL,
                            btNode<Item>* init_parent = NULL);
              btNode(const btNode<Item>* root_Ptr);
              ~btNode(); /*Deconstructor*/
              void operator=(const btNode<Item>* root_ptr);
              void set_data(const Item& entry){ data_field = entry;}
              void set_left_ptr(btNode<Item>* new_left){left_ptr = new_left;}
              void set_right_ptr(btNode<Item>* new_right){right_ptr = new_right;}
              void set_parent(btNode<Item>* new_parent){parent_ptr = new_parent;}
              bool isLeaf() const{
                     return (left_ptr == NULL) && (right_ptr == NULL);
              }
```

```
		Item data(){return data_field;}
		const Item data() const{return data_field;}
		btNode<Item>* left() { return left_ptr;}
		btNode<Item>* right() { return right_ptr;}
		btNode<Item>* parent() { return parent_ptr;}
		const btNode<Item>* left() const { return left_ptr;}
		const btNode<Item>* right() const { return right_ptr;}
		const btNode<Item>* parent() const {return parent_ptr;}
	private:
		btNode<Item>* parent_ptr;
		btNode<Item>* left_ptr;
		btNode<Item>* right_ptr;
		Item data_field;
		size_t count;
};

template<class Item, class Process>
void inOrderTraversal(const btNode<Item>* root_ptr, Process f);

template<class Item, class Process>
void preOrderTraversal(const btNode<Item>* root_ptr, Process f);

template<class Item, class Process>
void postOrderTraversal(const btNode<Item>* root_ptr, Process f);

template<class Item, class Process>
void backWardInOrderTraversal(const btNode<Item>* root_ptr, Process f);

template<class Item>
void print(const btNode<Item> * root_ptr);

template<class Item>
btNode<Item>* copybT(const btNode<Item> * root_ptr);

template<class Item>
void clearbT(btNode<Item>*& root_ptr);

template<class Item>
size_t numNodesbT(const btNode<Item>* root_ptr);

template<class Item>
size_t heightbT(const btNode<Item>*root_ptr);

#include "Sefa_B_HW5_Q1.cpp"
#endif
```

Implementation File: Sefa_Bujar_HW5_Q1.cpp:

```
#ifndef __BTNODE_CPP__
#define __BTNODE_CPP__
#include "Sefa_B_HW5_Q1.h"

template<class Item>
btNode<Item>::btNode( const Item& init_data,
                                       btNode<Item> * init_left,
                                       btNode<Item> * init_right,
                                       btNode<Item> * init_parent){
        data_field = init_data;
        left_ptr = init_left;
        right_ptr = init_right;
        parent_ptr = init_parent;
}

template<class Item>
btNode<Item>::~btNode(){

        data_field = 0;

}

template<class Item>
void print(const btNode<Item>* root_ptr){
        if(root_ptr ==NULL ) return;
        cout << root_ptr->data() << endl;
}

template<class Item>
size_t numNodesbT(const btNode<Item>* root_ptr){
        if(root_ptr == NULL){
                return 0;
        }

        size_t count = 1;
        count = count + numNodesbT(root_ptr->left());
        count = count + numNodesbT(root_ptr->right());
        return count;
}

template<class Item>
size_t heightbT(const btNode<Item>* root_ptr){
        if(root_ptr==NULL){
                return 0;

        }
        size_t leftCount = 0;
        size_t rightCount = 0;
        leftCount = leftCount + 1 + heightbT(root_ptr->left());
        rightCount = rightCount + 1 + heightbT(root_ptr->right());
        if(leftCount > rightCount)
                return leftCount;
```

```
	return rightCount;
}

/* NOTE: (left) Root (right) */
template<class Item, class Process>
void inOrderTraversal(const btNode<Item>* root_ptr, Process f){
	if(root_ptr == NULL){
		return;
	}
	/* Go through left side first. */
	inOrderTraversal(root_ptr->left(), f);
	f(root_ptr);
	inOrderTraversal(root_ptr->right(),f);

}

template<class Item, class Process>
void preOrderTraversal(const btNode<Item>* root_ptr, Process f){
	if(root_ptr == NULL){
		return;
	}
	/* Go through root first. */
	f(root_ptr);
	preOrderTraversal(root_ptr->left(), f);
	preOrderTraversal(root_ptr->right(),f);

}

template<class Item, class Process>
void postOrderTraversal(const btNode<Item>* root_ptr, Process f){
	if(root_ptr == NULL){
		return;
	}
	/* Go through left side first. */
	postOrderTraversal(root_ptr->left(), f);
	postOrderTraversal(root_ptr->right(),f);
	f(root_ptr);
}

template<class Item, class Process>
void backWardInOrderTraversal(const btNode<Item>* root_ptr, Process f){
	if(root_ptr == NULL){
		return;
	}
	/* Go through right side first. */
	backWardInOrderTraversal(root_ptr->right(), f);
	f(root_ptr);
	backWardInOrderTraversal(root_ptr->left(),f);

}

template<class Item>
void clearbT(btNode<Item>*& root_ptr){
	/*If the current pointer is null, do nothing*/
```

```
        if(root_ptr == NULL) return;
        /*Iterate down to every single leaf and delete that pointer and reset it to null, and
set its data to 0.*/
        btNode<Item>* child;
        btNode<Item>* parent;
        child = root_ptr->left();
        clearbT(child); /*go all the way to left.*/
        child = root_ptr->right();
        clearbT(child); /*go all the way to the right.*/
        delete root_ptr;
}

template<class Item>
btNode<Item>* copybT(btNode<Item>* root_ptr){
        if(root_ptr == NULL){
                return NULL; /*Since our return type is a pointer we must return null*/
        }
        btNode<Item> * left_ptr;
        btNode<Item> * right_ptr;
        btNode<Item> * parent_ptr;
        left_ptr = copybT(root_ptr->left());
        right_ptr = copybT(root_ptr->right());
        parent_ptr = copybT(root_ptr->parent());
        return new btNode<Item>(root_ptr->data(), left_ptr, right_ptr, parent_ptr);
}

#endif
```

binaryTree - LinkedList:

Header File: Sefa_Bujar_HW5_Q2.h:

```
// FILE: btClass.h
// TEMPLATE CLASS PROVIDED: binaryTree<Item> (a binary tree where each node has
//   an item) The template parameter, Item, is the data type of the items in the
//   tree's nodes. It may be any of the C++ built-in types (int, char, etc.),
//   or a class with a default constructor, a copy constructor and an assignment
//   operator.
//
// NOTE: Each non-empty tree always has a "current node."  The location of
// the current node is controlled by three member functions: shiftUp,
// shiftToRoot, shiftLeft, and shiftRight.
//
// CONSTRUCTOR for the binaryTree<Item> template class:
//   binaryTree( )
//     Postcondition: The binary tree has been initialized as an empty tree
//     (with no nodes).
//
// MODIFICATION MEMBER FUNCTIONS for the binaryTree<Item> template class:
//   void createFirstNode(const Item& entry)
//     Precondition: size( ) is zero.
//     Postcondition: The tree now has one node (a root node), containing the
//     specified entry. This new root node is the "current node."
//
//   void shiftToRoot( )
//     Precondition: size( ) > 0.
//     Postcondition: The "current node" is now the root of the tree.
//
//   void shiftUp( )
//     Precondition: hasParent( ) returns true.
//     Postcondition: The "current node" has been shifted up to the parent of
//     the old current node.
//
//   void shiftLeft( )
//     Precondition: hasLeft( ) returns true.
//     Postcondition: The "current node" been shifted down to the left child
//     of the original current node.
//
//   void shiftRight( )
//     Precondition: hasRight( ) returns true.
//     Postcondition: The "current node" been shifted down to the right child
//     of the original current node.
//
//   void change(const Item& new_entry)
//     Precondition: size( ) > 0.
//     Postcondition: The data at the "current node" has been changed to the
//     new entry.
//
```

```
//   void addLeft(const Item& entry)
//     Precondition: size( ) > 0, and hasLeft( ) returns false.
//     Postcondition: A left child has been added to the "current node,"
//     with the given entry.
//
//   void addRight(const Item& entry)
//     Precondition: size( ) > 0, and hasRight( ) returns false.
//     Postcondition: A right child has been added to the "current node,"
//     with the given entry.
//
// CONSTANT MEMBER FUNCTIONS for the binaryTree<Item> template class:
//   size_t size( ) const
//     Postcondition: The return value is the number of nodes in the tree.
//
//   Item retrieve( ) const
//     Precondition: size( ) > 0.
//     Postcondition: The return value is the data from the "current node."
//
//   bool hasParent( ) const
//     Postcondition: Returns true if size( ) > 0, and the "current node"
//     has a parent.
//
//   bool hasLeft( ) const
//     Postcondition: Returns true if size( ) > 0, and the "current node"
//     has a left child.
//
//   bool hasRight( ) const
//     Postcondition: Returns true if size( ) > 0, and the "current node"
//     has a right child.
//
// VALUE SEMANTICS for the binaryTree<Item> template class:
//   Assignments and the copy constructor may be used with binaryTree objects.
//
// DYNAMIC MEMORY USAGE by the binaryTree<Item> template class:
//   If there is insufficient dynamic memory, then the following functions
//   throw bad_alloc:
//   createFirstNode, addLeft, addRight, the copy constructor, and the
//   assignment operator.

#ifndef __BT_CLASS_H__
#define __BT_CLASS_H__


#include <cstdlib>    // Provides size_t
#include "Sefa_B_HW5_Q1.h"   // Provides btNode<Item> template class

using namespace std;

template <class Item>
class binaryTree
{
public:
    // CONSTRUCTORS and DESTRUCTOR
    binaryTree( );
    binaryTree(const binaryTree& source);
    ~binaryTree( );
    // MODIFICATION MEMBER FUNCTIONS
    void createFirstNode(const Item& entry);
    void shiftToRoot( );
    void shiftUp( );
    void shiftLeft( );
    void shiftRight( );
    void change(const Item& new_entry);
    void addLeft(const Item& entry);
    void addRight(const Item& entry);
    // CONSTANT MEMBER FUNCTIONS
    size_t size( ) const;
    Item retrieve( ) const;
       btNode<Item>* retrieveNode(); /* retrieve pointer to currentNode.*/
    bool hasParent( ) const;
    bool hasLeft( ) const;
    bool hasRight( ) const;
       void print(); /*Function I added just to test that everything is okay.*/
       void setRoot(btNode<Item>* new_root); /*This is a function that is needed for Q4 for
transplant. Tree may need to set its root to a tree.*/

private:
       btNode<Item> * currentNode;
       btNode<Item> * rootNode;
       size_t count;

};

#include "Sefa_B_HW5_Q2.cpp"
#endif
```

<u>Implementation File: </u>Sefa_Bujar_HW5_Q2.cpp:

```
#ifndef __BT_CLASS_CPP__
#define __BT_CLASS_CPP__

#include "Sefa_B_HW5_Q2.h"
#include "Sefa_B_HW5_Q1.h"

template<class Item>
binaryTree<Item>::binaryTree(){
	currentNode = NULL;
	rootNode = NULL;
	count = 0;
}

template<class Item>
binaryTree<Item>::~binaryTree(){
	clearbT(rootNode);
	delete currentNode;
	currentNode = NULL;
	count = 0;
}

template<class Item>
binaryTree<Item>::binaryTree(const binaryTree& source){
	rootNode = copybT(source);
	currentNode = rootNode;
	count = 0;
}

template<class Item>
void binaryTree<Item>::createFirstNode(const Item& entry){
	assert((currentNode == NULL) && (rootNode == NULL));
	currentNode = new btNode<Item>(entry);
	rootNode = currentNode;
	count = 1;
}

template<class Item>
bool binaryTree<Item>::hasParent() const{
	return (currentNode->parent() != NULL) && size()>0;
}

template<class Item>
bool binaryTree<Item>::hasLeft() const {
	return ((currentNode->left() != NULL) && (size()>0));
}

template<class Item>
bool binaryTree<Item>::hasRight() const{
	return ((currentNode->right() != NULL) && (size() > 0));
}
```

```
template<class Item>
size_t binaryTree<Item>::size() const{
	return count;
}

template<class Item>
Item binaryTree<Item>::retrieve() const{
	assert(this->size()>0);
	return currentNode->data();
}

template<class Item>
btNode<Item>* binaryTree<Item>::retrieveNode(){
	assert(this->size()>0);
	return currentNode;
}

template<class Item>
void binaryTree<Item>::addRight(const Item& entry){
	assert((size() >0) && (!hasRight()));
	btNode<Item> * newRight = new btNode<Item>(entry, NULL, NULL, currentNode);
	currentNode->set_right_ptr(newRight);
	count++;
}

template<class Item>
void binaryTree<Item>::addLeft(const Item& entry){
	assert((size()>0) && (!hasLeft()));
	btNode<Item> * newLeft = new btNode<Item>(entry,NULL,NULL, currentNode);
	currentNode->set_left_ptr(newLeft);
	count++;
}

/*template<class Item>
void binaryTree<Item>

*/
template<class Item>
void binaryTree<Item>::shiftLeft(){
	assert(hasLeft() && (size()>0));
	currentNode = currentNode->left();
}

template<class Item>
void binaryTree<Item>::shiftRight(){
	assert(hasRight() && (size() > 0));
	currentNode = currentNode->right();
}

template<class Item>
void binaryTree<Item>::change(const Item& entry){
	assert((currentNode !=NULL) && (size() > 0)); /*Make sure we have a currentNode*/
	currentNode->set_data(entry); /*Change entry as needed.*/
}

template<class Item>
void binaryTree<Item>::shiftToRoot(){
	assert(size()>0);
	currentNode = rootNode;
}

template<class Item>
void binaryTree<Item>::shiftUp(){
	assert(size()>0 && hasParent());
	currentNode = currentNode->parent();
}

template<class Item>
void binaryTree<Item>::print(){
	assert(currentNode!=NULL);
	cout << currentNode->data() << endl;
}

template<class Item>
void binaryTree<Item>::setRoot(btNode<Item> * new_root){
	if(new_root == rootNode){
		return;
	}
	clearbT(rootNode);
	rootNode = new_root;
	currentNode = rootNode;
}

#endif
```

binaryTree - Arrays:
Header File: Sefa_Bujar_HW5_Q3.h:

```
// FILE: btClass.h
// TEMPLATE CLASS PROVIDED: binaryTree<Item> (a binary tree where each node has
//   an item) The template parameter, Item, is the data type of the items in the
//   tree's nodes. It may be any of the C++ built-in types (int, char, etc.),
//   or a class with a default constructor, a copy constructor and an assignment
//   operator.
//
// NOTE: Each non-empty tree always has a "current node."  The location of
// the current node is controlled by three member functions: shiftUp,
// shiftToRoot, shiftLeft, and shiftRight.
//
// CONSTRUCTOR for the binaryTree<Item> template class:
//   binaryTree( )
//     Postcondition: The binary tree has been initialized as an empty tree
//     (with no nodes).
//

// MODIFICATION MEMBER FUNCTIONS for the binaryTree<Item> template class:
//   void createFirstNode(const Item& entry)
//     Precondition: size( ) is zero.
//     Postcondition: The tree now has one node (a root node), containing the
//     specified entry. This new root node is the "current node."
//
//   void shiftToRoot( )
//     Precondition: size( ) > 0.
//     Postcondition: The "current node" is now the root of the tree.
//
//   void shiftUp( )
//     Precondition: hasParent( ) returns true.
//     Postcondition: The "current node" has been shifted up to the parent of
//     the old current node.
//
//   void shiftLeft( )
//     Precondition: hasLeft( ) returns true.
//     Postcondition: The "current node" been shifted down to the left child
//     of the original current node.
//
//   void shiftRight( )
//     Precondition: hasRight( ) returns true.
//     Postcondition: The "current node" been shifted down to the right child
//     of the original current node.
//
//   void change(const Item& new_entry)
//     Precondition: size( ) > 0.
//     Postcondition: The data at the "current node" has been changed to the
//     new entry.
//
```

```
//   void addLeft(const Item& entry)
//     Precondition: size( ) > 0, and hasLeft( ) returns false.
//     Postcondition: A left child has been added to the "current node,"
//     with the given entry.
//
//   void addRight(const Item& entry)
//     Precondition: size( ) > 0, and hasRight( ) returns false.
//     Postcondition: A right child has been added to the "current node,"
//     with the given entry.
//
// CONSTANT MEMBER FUNCTIONS for the binaryTree<Item> template class:
//   size_t size( ) const
//     Postcondition: The return value is the number of nodes in the tree.
//
//   Item retrieve( ) const
//     Precondition: size( ) > 0.
//     Postcondition: The return value is the data from the "current node."
//
//   bool hasParent( ) const
//     Postcondition: Returns true if size( ) > 0, and the "current node"
//     has a parent.
//
//   bool hasLeft( ) const
//     Postcondition: Returns true if size( ) > 0, and the "current node"
//     has a left child.
//
//   bool hasRight( ) const
//     Postcondition: Returns true if size( ) > 0, and the "current node"
//     has a right child.
//
// VALUE SEMANTICS for the binaryTree<Item> template class:
//   Assignments and the copy constructor may be used with binaryTree objects.
//
// DYNAMIC MEMORY USAGE by the binaryTree<Item> template class:
//   If there is insufficient dynamic memory, then the following functions
//   throw bad_alloc:
//   createFirstNode, addLeft, addRight, the copy constructor, and the
//   assignment operator.

#ifndef __BT_CLASS_H__
#define __BT_CLASS_H__

#include <cstdlib>    // Provides size_t
#include "Sefa_B_HW5_Q1.h"   // Provides btNode<Item> template class

using namespace std;

template <class Item>
class binaryTree
{
public:
    // CONSTRUCTORS and DESTRUCTOR
    binaryTree(size_t init_cap = 2);
    binaryTree(const binaryTree& source);
    ~binaryTree( );
    // MODIFICATION MEMBER FUNCTIONS
    void createFirstNode(const Item& entry);
    void shiftToRoot( );
    void shiftUp( );
    void shiftLeft( );
    void shiftRight( );
    void change(const Item& new_entry);
    void addLeft(const Item& entry);
    void addRight(const Item& entry);
    // CONSTANT MEMBER FUNCTIONS
    size_t size( ) const;
    Item retrieve( ) const;
    bool hasParent( ) const;
    bool hasLeft( ) const;
    bool hasRight( ) const;
       void resize(const size_t& newSize);
private:
       Item* treeData;
       size_t CAPACITY;
       size_t count;
       size_t currentIndex;
};

#include "Sefa_B_HW5_Q3.cpp"
#endif
```

<u>Implementation File: </u>Sefa_Bujar_HW5_Q3.cpp:

```
#ifndef __BT_CLASS_CPP__
#define __BT_CLASS_CPP__


#include "Sefa_B_HW5_Q3.h"

template<class Item>
binaryTree<Item>::binaryTree(size_t init_cap){
	CAPACITY = init_cap;
	treeData = new Item[CAPACITY];
}

template<class Item>
binaryTree<Item>::binaryTree(const binaryTree& source){
	CAPACITY = source.CAPACITY;
	count = source.count;
	currentIndex = source.currentIndex;
	treeData = new Item[CAPACITY];
	for(size_t i = 0; i < CAPACITY; i++){
		treeData[i] = source.treeData[i];
	}
}

template<class Item>
void binaryTree<Item>::createFirstNode(const Item& entry){
	count = 1;
	currentIndex = 0;
	treeData[currentIndex] = entry;
}

template<class Item>
binaryTree<Item>::~binaryTree(){
	delete [] treeData;
	CAPACITY = 0;
	count = 0;
	currentIndex = -1;
}

template<class Item>
size_t binaryTree<Item>::size() const{
	return count;
}

template<class Item>
Item binaryTree<Item>::retrieve( ) const{
	assert((size() > 0) && ((currentIndex >= 0) && (currentIndex < CAPACITY)));
	return treeData[currentIndex];
}

template<class Item>
void binaryTree<Item>::shiftToRoot(){
	assert((size() > 0) && ((currentIndex >= 0) && (currentIndex < CAPACITY)));
```

```
        currentIndex = 0;
}

template<class Item>
bool binaryTree<Item>::hasParent() const{
        assert((size() > 0) && ((currentIndex >= 0) && (currentIndex < CAPACITY)));
        return currentIndex != 0;
}

template<class Item>
void binaryTree<Item>::shiftUp(){
        assert(((size() > 0) && ((currentIndex >= 0) && (currentIndex < CAPACITY))) &&
hasParent());
        currentIndex = (currentIndex-1)/2;
}


/*NOTE NOTE NOTE might need to make <= count... but count is always +1 so no. Just incase u
have an error it could be at these two...*/
template<class Item>
bool binaryTree<Item>::hasLeft() const{
        assert((size() > 0) && ((currentIndex >= 0) && (currentIndex < CAPACITY)));
        return (2*currentIndex+1) < size();
}

template<class Item>
bool binaryTree<Item>::hasRight() const{
        assert((size() > 0) && ((currentIndex >= 0) && (currentIndex < CAPACITY)));
        return (2*currentIndex+2) < size();
}

template<class Item>
void binaryTree<Item>::shiftLeft(){
        assert(size() > 0 && hasLeft());
        currentIndex = 2*currentIndex + 1;
}

template<class Item>
void binaryTree<Item>::shiftRight(){
        assert(size() > 0 && hasRight());
        currentIndex = 2*currentIndex + 2;
}

template<class Item>
void binaryTree<Item>::addLeft(const Item& entry){
        assert((size() > 0 && !hasLeft())); //&& (size() < CAPACITY));
        if(!(size()<CAPACITY)){
                resize(CAPACITY*2);
        }
        treeData[(2*currentIndex+1)] = entry;
        count++;
}

template<class Item>
```

```
void binaryTree<Item>::addRight(const Item& entry){
	assert((size() > 0 && !hasRight()));//&& (size() <CAPACITY));
	if(!(size()<CAPACITY)){
		resize(CAPACITY*2);
	}
	treeData[(2*currentIndex+2)] = entry;
	count++;
	/*cout << "HERE";*/
}

template<class Item>
void binaryTree<Item>::change(const Item& new_entry){
	assert(size() > 0);
	treeData[currentIndex] = new_entry;
}


template<class Item>
void binaryTree<Item>::resize(const size_t& new_size){
	if(new_size == CAPACITY)
		return;
	if(new_size < count)
		return;
	Item * newTree = new Item[new_size];
	for(size_t i = 0; i < count; i++){
		newTree[i] = treeData[i];
	}
	delete treeData;
	CAPACITY = new_size;
	treeData = newTree;
}

#endif
```

<u>Binary Search Tree</u>:
<u>Header File: </u>Sefa_Bujar_HW5_Q4.h:

```
#ifndef __BST_CLASS_H__
#define __BST_CLASS_H__

#include "Sefa_B_HW5_Q2.h"
#include <cstdlib> //Provides size_t


template<class Item>
class binarySearchTree{
	public:
		binarySearchTree(const Item& entry);
		void addNode(const Item& entry);
		void removeNode(btNode<Item> * nodeZ);
		btNode<Item>* searchNode(const Item& target);
		btNode<Item>* minimum(btNode<Item>* node);
		btNode<Item>* maximum(btNode<Item>* node);

		void printP();
		void transplant(btNode<Item>* oldTree, btNode<Item>* newTree);

		void operator=(const binarySearchTree<Item>& source);
		void printT();
	private:
		binaryTree<Item> tree_ptr;
		/*Don't need this, but going to add it just to have here*/
		size_t count;
};

#include "Sefa_B_HW5_Q4.cpp"

#endif
```

<u>Implementation File: </u>Sefa_Bujar_HW5_Q4.cpp:

```
#ifndef __BST_CLASS_CPP__
#define __BST_CLASS_CPP__

#include "Sefa_B_HW5_Q4.h"

template<class Item>
binarySearchTree<Item>::binarySearchTree(const Item& entry){
	tree_ptr = binaryTree<Item>();
	tree_ptr.createFirstNode(entry);
	count = 1;
}

template<class Item>
btNode<Item>* binarySearchTree<Item>::searchNode(const Item& target){
	btNode<Item> * retNode = NULL;
	if(target == tree_ptr.retrieve()){
		retNode = tree_ptr.retrieveNode();
		return retNode;
	}
	bool found = false;
	while(!found){
		if(target == tree_ptr.retrieve()){
			retNode = tree_ptr.retrieveNode();
			found = true;
		}
		else if(target < tree_ptr.retrieve() && tree_ptr.hasLeft()){
			tree_ptr.shiftLeft();
		}
		else if(target > tree_ptr.retrieve() && tree_ptr.hasRight()){
			tree_ptr.shiftRight();
		}
		else{
			found = true;
		}
	}
	tree_ptr.shiftToRoot();
	return retNode;
}

template<class Item>
void binarySearchTree<Item>::addNode(const Item& entry){
	if(this == NULL){
		tree_ptr = binaryTree<Item>();
		tree_ptr.createFirstNode(entry);
		count = 1;
	}
	if((tree_ptr.size() == 0)){
		tree_ptr.createFirstNode(entry);
		return;
	}
	bool found = false;
	while(!found){
```

```
                if(entry <= tree_ptr.retrieve() && tree_ptr.hasLeft()){
                        tree_ptr.shiftLeft();
                }
                else if(entry > tree_ptr.retrieve() && tree_ptr.hasRight()){
                        tree_ptr.shiftRight();
                }
                else{
                        found = true;
                }
        }
        if(entry <= tree_ptr.retrieve()){
                tree_ptr.addLeft(entry);
        }
        else{
                tree_ptr.addRight(entry);
        }
        tree_ptr.shiftToRoot();

}

template<class Item>
btNode<Item>* binarySearchTree<Item>::minimum(btNode<Item>* node){
        assert(node!=NULL);
        btNode<Item>* found = searchNode(node->data());
        assert(found !=NULL);
        btNode<Item>* minPtr = node;
        btNode<Item>* iterator = node;
        while(iterator->left() !=NULL){
                //minPtr = iterator;
                iterator = iterator->left();
                minPtr = iterator;
        }
        return minPtr;
}

template<class Item>
btNode<Item>* binarySearchTree<Item>::maximum(btNode<Item>* node){
        assert(node!=NULL);
        btNode<Item>* found = searchNode(node->data());
        assert(found !=NULL);
        btNode<Item>* maxPtr = node;
        btNode<Item>* iterator = node;
        while(iterator->right() !=NULL){
                //maxPtr = iterator;
                iterator = iterator->right();
                maxPtr = iterator;
        }
        return maxPtr;
}
```

```
template<class Item>
void binarySearchTree<Item>::transplant(btNode<Item>* oldTree,
                                     btNode<Item>* newTree){
       if(oldTree->parent() == NULL){
              tree_ptr.setRoot(newTree);
       }
       else if(oldTree == (oldTree->parent())->left()){
              (oldTree->parent())->set_left_ptr(newTree);
       }
       else{
              (oldTree->parent())->set_right_ptr(newTree);
       }
       if(newTree!=NULL){
              newTree->set_parent(oldTree->parent());
       }
}



template<class Item>
void binarySearchTree<Item>::removeNode(btNode<Item>* nodeZ){
       assert(nodeZ!=NULL);
       btNode<Item>* found = searchNode(nodeZ->data()); /*Make sure this node is in the tree*/
       assert(found !=NULL);
       if(nodeZ->left() == NULL && nodeZ->right() == NULL){
              (nodeZ->parent())->set_parent(NULL);
              nodeZ->set_data(0);
       }
       if(nodeZ->left() == NULL){
              transplant(nodeZ, nodeZ->right());
       }
       else if(nodeZ->right() == NULL){ /*else if only left, shift to current*/
              transplant(nodeZ, nodeZ->left());
       }
       else{
              /*If we have two children, here is the hard part. We must first find the new
smallest maximum, or sucessor */
              btNode<Item>* nodeZSuccessor = minimum(nodeZ->right());
              if(nodeZSuccessor->parent() != nodeZ){
                     transplant(nodeZSuccessor, nodeZSuccessor->right());
                     nodeZSuccessor->set_right_ptr(nodeZ->right());
                     (nodeZSuccessor->right())->set_parent(nodeZSuccessor);
              }
              transplant(nodeZ, nodeZSuccessor);
              nodeZSuccessor->set_left_ptr(nodeZ->left());
              (nodeZSuccessor->left())->set_parent(nodeZSuccessor);
       }

}
```

```
template<class Item>
void binarySearchTree<Item>::printT(){
	tree_ptr.print();
	cout << "SIZE " <<tree_ptr.size()<< endl;
	tree_ptr.shiftRight();
	tree_ptr.print();

	cout << "Size " << tree_ptr.size() << endl;
	tree_ptr.shiftRight();
	tree_ptr.print();

	/*Send it all the way back up or it will mess everything up.*/
	tree_ptr.shiftToRoot();
}



template<class Item>
void binarySearchTree<Item>::printP(){
	btNode<Item> * bt = searchNode(100);
	inOrderTraversal(bt, print<int>);

}
#endif
```

<u>Heap</u>:

<u>Header File: </u>Sefa_Bujar_HW5_Q5.h:

```
#ifndef __HEAP_CLASS_H__
#define __HEAP_CLASS_H__

#include "Sefa_B_HW5_Q3.h"
#include <cstdlib> //Provides size_t

template<class Item>
class heap{
	public:
		heap(size_t init_capcity = 30);

		/*Big 3 have not been implemented.*/
		~heap();
		heap(const heap<Item>& source);
		void operator=(const heap<Item>& source);

		void addNode(const Item& entry);
		size_t deleteNode(); /*Just deletes the highest node, the root node. */
		size_t searchNode(const Item& target);
		/*Note here I am just returning index of where the node would be*/
		size_t minimum();
		size_t maximum();
		void resize(const size_t& new_size);
		void print();
		size_t root(); /*NOTE, I just wrote this haven't used it.*/
	private:
		/*binaryTree<int> * heapData;*/
		/*size_t capacity; */
		/*NOTE binaryTree handles all the size etc and asserting, etc.*/
		Item * heapData;
		size_t CAPACITY;
		size_t count;

};

#include "Sefa_B_HW5_Q5.cpp"

#endif
```

<u>Implementation File: </u>Sefa_Bujar_HW5_Q5.cpp:

```
#ifndef __HEAP_CLASS_CPP__
#define __HEAP_CLASS_CPP__

#include "Sefa_B_HW5_Q5.h"

template<class Item>
heap<Item>::heap(size_t init_cap){
	heapData = new Item[init_cap];
	CAPACITY = init_cap;
	count = 0;
}

template<class Item>
void heap<Item>::addNode(const Item& entry){
	if(!(count<CAPACITY)){
		resize(CAPACITY*2);
	}
	heapData[count++] = entry; /*Count is incremented. */
	size_t i = count-1; /*Set i to the index of the last, will be used to keep track of 
which elements to swap*/
	while(i!=0 && (heapData[i]> heapData[(i-1)/2])){
			Item temp = heapData[i];
			heapData[i] = heapData[(i-1)/2];
			heapData[(i-1)/2] = temp;
			i = (i-1)/2; /*Keep swapping with parent until it gets to the spot it 
needs to.*/
	}
}


template<class Item>
size_t heap<Item>::deleteNode(){
	assert(count > 0);
	size_t retVal = heapData[0];
	/*cout << endl;
	cout << "RETVAL " << retVal<<endl;
	cout << "LAST" << heapData[count-1] << endl;
	cout << endl;*/
	heapData[0] = heapData[--count];
	if(count == 1){
		return retVal;
	}
	size_t i = 0; /*Set index to start*/
	while((i < count) && (2*i+1)<count){
		Item temp = heapData[i];
		/*NOTE it must be swapped with the larger child. I will have it go to the left 
if the two chilren are equal.*/
		if(!(2*i+2<count))
			i = count;
		else if((heapData[i] < heapData[2*i+1]) && (heapData[2*i+1] >= 
heapData[2*i+2])){
			heapData[i] = heapData[2*i+1];
```

```
				heapData[2*i+1] = temp;
				i = 2*i+1;
			}
			else if((heapData[i]< heapData[2*i+2]) && (heapData[2*i+2] > heapData[2*i+1])){
				heapData[i] = heapData[2*i+2];
				heapData[2*i+2] = temp;
				i = 2*i+2;
			}
			else{
				i = count;
			}

			/*if((2*i+1) >= count && (2*i+2) >= count){
				i = count;
			}*/
		}
		return retVal;
}

template<class Item>
void heap<Item>::resize(const size_t& new_size){
		if(new_size == CAPACITY)
			return; /*This is saying, if what we want is already the same, dont do
anything.*/
		if(new_size < count)
			return; /*This is saying that if we want something smaller than the entries,
dont do it becuase we cant lose stuff.*/
		Item * newHeap = new Item[new_size];
		for(size_t i = 0; i < count; i++){
			newHeap[i] = heapData[i];
		}
		delete heapData;
		CAPACITY = new_size;
		heapData = newHeap;
}


template<class Item>
size_t heap<Item>::minimum(){
		assert(count > 0);
		size_t min = heapData[0];
		if(count == 1){
			return min;
		}
		for(size_t i = 1; i < count; i++){
			if(heapData[i] < min){
				min = heapData[i];
			}
		}
		return min;
}

template<class Item>
size_t heap<Item>::maximum(){
        assert(count > 0);
        size_t max = heapData[0];
        if(count == 1){
                return max;
        }
        for(size_t i = 1; i < count; i++){
                if(heapData[i] > max){
                        max = heapData[i];
                }
        }
        return max;
}

template<class Item>
size_t heap<Item>::root(){
        assert(count>0);
        return heapData[0];

}

template<class Item>
void heap<Item>::print(){
        for(size_t i = 0; i < count; i++){
                cout << heapData[i] << " ";
        }
        cout <<endl;

}


#endif
```

<u>HeapSort</u>:

<u>Header File: </u>Sefa_Bujar_HW5_Q5b.h:

```
#include <cstdlib>
#include <cassert>
#include <iostream>


using namespace std;
void heapSort(int arr[], size_t size);
void reheapify_down(int arr[], size_t size);
void makeHeap(int arr[], size_t size);
```

<u>Implementation File:</u> Sefa_Bujar_HW5_Q5b.cpp:

```
#include "Sefa_B_HW5_Q5b.h"

void makeHeap(int arr[], size_t size){
	/*NOTE this is literally like add. The way the book is doing it is that it is teating 
heapsort as its own thing. Meaning that it just changes the arary thats given instead of 
actually making a heap as an array.*/
	size_t indexNew;
	for(size_t i = 1; i < size; i++){
		indexNew = i;
		while((indexNew>0) && (arr[indexNew] > arr[(indexNew-1)/2])){
			int temp = arr[indexNew];
			arr[indexNew] = arr[(indexNew-1)/2];
			arr[(indexNew-1)/2] = temp;
			indexNew = (indexNew-1)/2;
		}
	}
}

/*This function is basically like delete, just moves larger child up, and root down.*/
void reheapify_down(int arr[], size_t size){
	size_t i = 0;
	if(size == 1){
		return;
	}
	/*While we are not a heap, and we have a child to swap with.*/
	while((i<size) && (2*i+1)<size){
		int temp = arr[i];
		if((2*i+2) >= size){ /*NOTE if our right side is too big, stop this loop.*/
			i = size;
		}
		else if((arr[i] < arr[2*i+1])&&(arr[(2*i+1)]>=arr[2*i+2])){
			arr[i] = arr[2*i+1];
			arr[2*i+1] = temp;
			i = 2*i+1;
		}
		else if((arr[i]< arr[2*i+2]) && (arr[2*i+2] > arr[2*i+1])){
			arr[i] = arr[2*i+2];
			arr[2*i+2] = temp;
			i = 2*i+2;
		}
		else{
			i = size;
		}
	}

}

void heapSort(int arr[], size_t size){
	size_t i;
	makeHeap(arr, size); /*Turn array into heap.*/
	i = size;
	while(i > 1){
```

```
            --i; /*Decrease index so its valid. Start from last.*/
            /*Swap root with last. */
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;
            reheapify_down(arr, i);
        }

}
```